The theory behind the IHttpFilter interface

devblogs.microsoft.com/oldnewthing/20231117-00

November 17, 2023

Raymond Chen

The Windows Runtime has an interface called `IHttpFilter` that lets you customize how HTTP requests are processed. The system has a default implementation called `HttpBaseProtocolFilter`, and if it has the features you need, then you're all set:

```
auto filter = HttpBaseProtocolFilter();
filter.AllowAutoRedirect(false); // disable auto-redirect
auto client = HttpClient(filter);
⟦ Use the client ⟧
```

But maybe you want to do something beyond that. For example, maybe you want a filter that injects an extra header into each request. You can implement these extra features by providing your own implementation of `IHttpFilter`, using the default implementation to do the heavy lifting.

```
// IDL
runtimeclass ExtraHeadersHttpFilter : Windows.Web.Http.Filters.IHttpFilter
{
    ExtraHeadersHttpFilter();
    void AddHeader(string name, string value);
}

// Implementation

struct ExtraHeadersHttpFilter : ExtraHeadersHttpFilterT<ExtraHeadersHttpFilter>
{
    using namespace Http = winrt::Windows::Web::Http;
private:
    // Data members
    std::mutex m_mutex;
    std::unordered_map<winrt::hstring, winrt::hstring> m_extraHeaders;
    Http::Filters::HttpBaseProtocolFilter m_base =
Http:Filters::HttpBaseProtocolFilter();

public:
    ExtraHeadersHttpFilter() = default;

    // Bonus methods for public consumption
    void AddHeader(winrt::hstring const& name,
                   winrt::hstring const& value)
    {
        auto lock = std::lock_guard(m_mutex);
        m_extraHeaders.insert_or_assign(name, value);
    }

    // IHttpFilter methods

winrt::Windows::Foundation::IAsyncOperationWithProgress<Http::HttpResponseMessage,
        Http::HttpProgress> SendRequestAsync(Http::HttpRequestMessage request)
    {
        // Add our bonus headers to the request
        {
            auto lock = std::lock_guard(m_mutex);
            auto headers = request.Headers();
            for (auto [name, value] : m_extraHeaders) {
                headers.Insert(name, value);
            }
        }

        // And then use default handling for the rest.
        return m_base.SendRequestAsync(request);
    }
};

// Consumer
auto filter = ExtraHeadersHttpFilter();
filter.AddHeader(L"X-Contoso", L"Awesome");
```

```
auto client = HttpClient(filter);
⟦ Use the client ⟧
```

One thing that people like to do is stack filters on top of each other. Maybe you want to use the `ExtraHeadersHttpFilter` in conjunction with a `CustomRetryHttpFilter`. Right now, the `ExtraHeadersHttpFilter` hands all of its requests to the default `HttpBaseProtocolFilter`, so there’s no way to combine it with a `CustomRetryHttpFilter`.

The way to fix this is to give the `ExtraHeadersHttpFilter` a constructor that takes another filter. Requests then pass through that custom filter instead of going to the default filter.

```
// IDL
runtimeclass ExtraHeadersHttpFilter : Windows.Web.Http.Filters.IHttpFilter
{
    ExtraHeadersHttpFilter();
    ExtraHeadersHttpFilter(Windows.Web.Http.Filters.IHttpFilter baseFilter);
    void AddHeader(string name, string value);
}

// Implementation

struct ExtraHeadersHttpFilter : ExtraHeadersHttpFilterT<ExtraHeadersHttpFilter>
{
    using namespace Http = winrt::Windows::Web::Http;
private:
    // Data members
    std::mutex m_mutex;
    std::unordered_map<winrt::hstring, winrt::hstring> m_extraHeaders;
    Http:Filters::IHttpFilter m_base = Http:Filters::HttpBaseProtocolFilter();

public:
    ExtraHeadersHttpFilter() = default;

    ExtraHeadersHttpFilter(Http::Filters::IHttpFilter const& baseFilter) :
        m_base(baseFilter) {}

    // Bonus methods for public consumption
    void AddHeader(winrt::hstring const& name,
                   winrt::hstring const& value)
    {
        auto lock = std::lock_guard(m_mutex);
        m_extraHeaders.insert_or_assign(name, value);
    }

    // IHttpFilter methods

winrt::Windows::Foundation::IAsyncOperationWithProgress<Http::HttpResponseMessage,
        Http::HttpProgress> SendRequestAsync(Http::HttpRequestMessage request)
    {
        // Add our bonus headers to the request
        {
            auto lock = std::lock_guard(m_mutex);
            auto headers = request.Headers();
            for (auto [name, value] : m_extraHeaders) {
                headers.Insert(name, value);
            }
        }

        // And then pass the request to the base filter.
        return m_base.SendRequestAsync(request);
    }
};
```

```
// Consumer
auto retryFilter = CustomRetryHttpFilter(3);
auto headerFilter = ExtraHeadersHttpFilter(retryFilter);
headerFilter.AddHeader(L"X-Contoso", L"Awesome");
auto client = HttpClient(headerFilter);
⟦ Use the client ⟧
```

The idea behind HTTP filters is that each request passes through the filters before hitting the wire, and then the response passes through the filters on the way back.

App		Filter1		Filter2				FilterN		Network
Request	→	•	→	•	→	…	→	•	→	↴
				↑		…				•
•	←	•	←	•	←	…	←	•	←	Response

In the above diagram, Filter2 is a Retry filter, so when it gets a response, it might decide to generate a new request and submit that one down the pipeline.

Basically, each filter can treat the remainder of the pipeline as a black box which conceptually submits the request and produces a result. Your filter doesn't even have to submit the request down the pipeline; maybe it sees something wrong with the request and chooses to generate an error response without actually hitting the wire. Or it might submit the request to two downstream pipelines for some reason. Or you may have a custom filter for testing purposes that replays prerecorded Web responses without every submitting them to the network at all. Each filter can decide whatever it likes.

Bonus chatter: For expository purposes, I didn't show how to modify the result. You can find an example of that in the HttpClient sample's PluginFilter filter.